Dynamic Energy Management of FPGA Accelerators in Embedded Systems

MOHAMMAD HOSSEINABADY, University of Bristol
JOSE LUIS NUNEZ-YANEZ, University of Bristol

In this paper, we investigate how to utilise an FPGA in an embedded system to save energy. For this purpose, we study the energy efficiency of a hybrid FPGA-CPU device that can switch task execution between hardware and software with focus on periodic tasks. To increase the applicability of this task switching, we also consider the voltage and frequency scaling (VFS) applied to the FPGA to reduce the system energy consumption. We show that in some cases, if the task's period is higher than a specific level, the FPGA accelerator cannot reduce the energy consumption associated to the task and the software version is the most energy efficient option. We have applied the proposed techniques to a robot map creation algorithm as a case study which shows up to 38% energy reduction compared to the FPGA implementation. Overall, experimental results show up to 48% energy reduction by applying the proposed techniques at runtime on thirteen individual tasks.

CCS Concepts: • **Hardware → Power and energy**; **Power estimation and optimization**; **Platform power issues**;

Additional Key Words and Phrases: FPGA, Dynamic Voltage and Frequency Scaling, Dynamic Power Management, Zynq-SoC, Dynamic Energy Management

ACM Reference format:
Mohammad Hosseinabady and Jose Luis Nunez-Yanez. 2017. Dynamic Energy Management of FPGA Accelerators in Embedded Systems. *ACM Trans. Web* 9, 4, Article 39 (March 2017), 25 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Embedded systems consisting of computation, communication and memory cores are used extensively in different devices such as mobile phones, smart cards, satellites, robots, medical monitoring devices, home appliances and so on. Most of these embedded systems perform periodic tasks during their life-time. For example, an autonomous mobile robot periodically monitors the environment by using different types of sensors to build a map of an unknown environment or to detect objects. In addition, most mobile embedded systems draw their power from batteries and utilise small computational resources. The limited *computational resources* and *energy budgets* require the consideration of specific design techniques for portable embedded systems. Therefore, researchers have proposed using FPGA fabrics along with main processors in such systems. Examples are

This work is supported by the Engineering and Physical Sciences Research Council, under grant EP/L00321X/1 (ENPOWER).
Author's addresses: M. Hosseinabady, Electrical and Electronic Engineering Department, University of Bristol, UK. Email: m.hosseinabady@bristol.ac.uk; J. L. Nunez-Yanez, Electrical and Electronic Engineering Department, University of Bristol, UK. Email: J.L.Nunez-Yanez@bristol.ac.uk.

1559-1131/2017/3-ART39 $15.00
https://doi.org/0000001.0000001

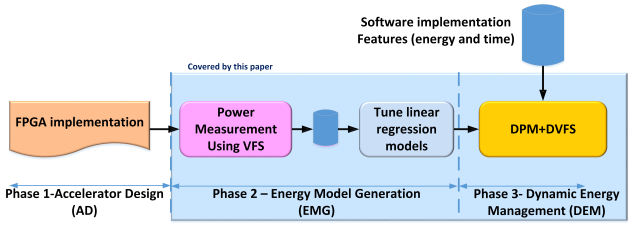


Fig. 1. DEM general view

Internet of Things (IoTs) devices and robots that usually utilise small FPGAs thanks to their size constraints. Because of limited resources available on these FPGAs, they can host a single task at a time. Consequently, when the FPGA is idle, especially in periodic tasks, its energy consumption to maintain the configuration can be significant. Addressing this issue is the main concern of this paper.

Traditionally, FPGAs are used as accelerators to improve the computational power by implementing compute-intensive tasks. Previous research has also reported the high performance/Watt factor of FPGAs compared to processors and GPUs. However, if energy is the optimization objective and not just power, it is not clear how FPGAs can be used to save energy in an embedded system. In this paper, we try to answer this question: "How can an FPGA added to an embedded system save energy when running a single periodic task?" In this context a periodic task is a task that executes repeatedly with a specific rate. For this purpose, we show that the FPGA idle energy consumption which is required for holding its configuration can increase the total energy consumption which in turn decreases the FPGA effectiveness. Therefore, we utilise three techniques to reduce the total energy consumption of a periodic task. For a short FPGA idle time, the first technique which is based on voltage and frequency scaling on the FPGA tries to increase the task execution time to reduce the idle time in favor of energy reduction [1]. If the idle time is still dominant then the second technique considers switching the task from FPGA to the CPU and completely shutting down the FPGA to reduce the energy. If the switching to the software version cannot reduce the energy consumption, then the third technique considers turning off the FPGA during the idle time.

A Dynamic Energy Management (DEM), depicted in Fig. 1, is proposed to implement these three techniques. It consists of three phases and the last two phases are the main focus of this paper. Using High-Level Synthesis (HLS) tools, the first phase of this flow, i.e., Accelerator Design (AD), provides a hardware implementation for a given task. The second phase is Energy Model Generation (EMG) during which numerous empirical measurements for different values of FPGA's voltage and frequency take place to be used for training regression models that represent the task execution time and energy consumption. Using these models, Phase 3 predicts the most energy efficient implementation of the task along with the power mode of the FPGA at runtime.

The novel contributions of this research are as follows:

- Focusing on periodic tasks, we will discuss the difficulties of utilising FPGAs to reduce the energy consumption in embedded system
- Considering the memory energy consumption as well as the FPGA and processors
- Proposing a linear model to describe the relation between the system energy consumption and the FPGA voltage to be used in a DVFS scheme
- Coupling DPM with DVFS to evaluate the proposed models and techniques
- Considering a wide range of tasks synthesized by an HLS tool to evaluate the proposed methodology

We have applied the proposed techniques to a robot map creation algorithm which shows up to 38% energy reduction compared to using only the FPGA implementation. Experimental results show up to 48% energy reduction by applying the proposed DEM at runtime on thirteen individual tasks.

The rest of this paper is organized as follows: The next section explains some definitions and assumptions required by the rest of this paper. Using a simple example, Section 3 explains the motivations behind this research. Section 4 reviews the previous work. Phase 2 of Fig. 1 is explained in Section 5 . Section 6 discusses Phase 3 of Fig. 1. Section 7 considers a robot map creation algorithm as a case study to apply the proposed technique on a practical situation. Section 8 shows the results of applying the proposed techniques to 13 micro-benchmarks. Finally, Section 9 concludes the paper.

2 PRELIMINARIES

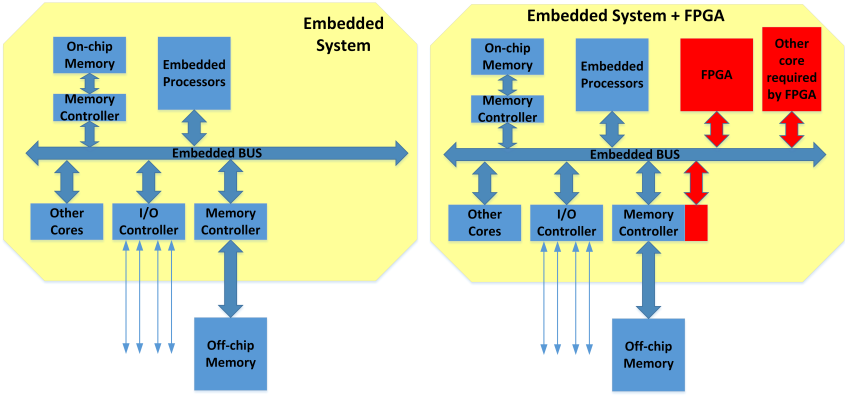
Before delving into the details of the proposed techniques, this section briefly explains a few concepts, definitions and assumptions considered in the rest of this paper.

A typical embedded system consists of computation, communication and memory cores as shown in Fig. 2. The processors usually run an operating system orchestrating all activities in the system. As mentioned before, our goal is adding an FPGA and its required cores to the system to be used as an accelerator and also save energy for some computations. Fig. 2b shows the modified system with the additional FPGA. The timeline of Fig. 3 shows a simplified model of the power consumption of different components during the life-time of the system. The power consumption of the processors and memory with only the operating system running, without any specific application, is called *baseline* power, denoted by the green colour in Fig 3. The power consumption of a software task running on the processors are added to the baseline power and is called *background* power from the point of view of other tasks running on the FPGA or CPU. This power is represented by blue colour in Fig 3.

Note that before assigning a task to the FPGA, the processor sends a few data values to the FPGA registers as the task arguments (denoted by te in Fig. 3) and after finishing the task the processor may receive a few data items as the return arguments (denoted by tp in Fig. 3). Therefore, the FPGA task power consumption, indicated by red colour in Fig 3, consists of the power of all cores incorporating the task execution which mainly includes the FPGA, the processor during tp and te and the main memory corresponding to the task memory access.

Modern commercial embedded FPGAs such as Xilinx Zynq SoC/MPSoC and Intel Cyclone V [2], utilise separate voltage rails for different sections in the FPGA, CPU and memory subsystems. These voltage rails can be set through regulators at the board level. The Intel Stratix 10 FPGA provides SmartVoltage ID control over VCC as its standard option that enables a smart voltage regulator to operate the device at lower VCC while maintaining performance [3]. Using these features, system level DVFS techniques are proposed for FPGAs [1, 4–7]. Hosseinabady et al. [8] explain how to scale the voltage of the core logic in the Xilinx Zynq SoC. They also calculate the timing overhead of scaling the voltage to shut down the FPGA core logic. A MicroBlaze-based light-weight soft-core IP is proposed by [4] to read and set the voltage lanes on the Zynq SoC through on-board voltage regulators supporting the PMBus protocol [9]. Using this IP, Nunez-Yanez et al. [1] propose a dynamic voltage and frequency scaling technique on the Xilinx Zynq to reduce the energy consumption of the motion estimation task.

Throughout this paper, we use the Xilinx Zynq ZC702 evaluation board and the DVFS library provided at [10]. The Xilinx Zynq SoC is a hybrid FPGA-CPU embedded systems based on the architecture shown in Fig 2b. This SoC consists of three main parts: Processing System (PS), Programmable Logic (PL) and Memory subsystem (MEM). The PS consists of a dual-core Cortex-A9



(a) Embedded system architecture without FPGA

(b) Embedded system architecture with FPGA

Fig. 2. Simplified embedded system architecture

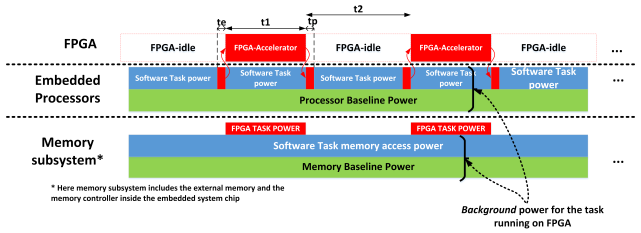


Fig. 3. Power consumption timeline

processor and NEON coprocessors associated to each core. The PL is an FPGA communicating to the PS and MEM using low performance and four non-cached and one cached high-performance ports, respectively. The MEM consists of off-chip DDR3 memory and on-chip memory controllers. Power consumptions of these three parts are measurable and changeable thanks to their separate voltage rails. The Xilinx Zynq ZC702 features XC7Z020 FPGA which is one the smallest FPGA in its family. This justifies its usage under single task accelerator assumption.

The Xilinx Vivado-HLS as a high-level synthesis tool which transforms a C/C++ code into the equivalent HDL code is used in this paper to implement FPGA-based accelerators. To get a high-performance from the accelerators, tasks transfer their data between the main memory and the FPGA internal memory (BRAM) using a burst data transfer mechanism that can be automatically generated by the Xilinx Vivado-HLS. The task may first transfer the data into the FPGA and then perform the computation or transfer the data while the computation performs in a streaming manner [11].

3 MOTIVATIONS AND PROBLEM FORMULATION

It is a general consensus that using FPGA as a computational resource can reduce the system energy consumption. Previous research [12, 13] has reported the FPGA energy efficiency considering one or more specific tasks under predefined situations. However, this could be incorrect especially for periodic tasks. This section first explains four pitfalls in measuring a task energy consumption

which can affect the energy reduction techniques. Then using the matrix-vector multiplication as an illustrative example, we show the impact of these pitfalls on the total energy consumption. Finally, these pitfalls motivate us to define potential opportunities to save energy in an FPGA-CPU hybrid embedded system, which leads us to propose a series of techniques to realise these potentials.

3.1 Pitfalls

The energy consumption of a task running on an embedded system is defined as the amount of energy that the task adds to the system. Whereas, software-implemented tasks increase the energy consumption on processors and memory subsystem (the corresponding power consumptions are shown in blue colour in Fig. 3), the task running on the FPGA increases the energy consumption on the FPGA, memory subsystem and processors (the corresponding power consumptions are shown in red colour in Fig. 3). Any misconception in measuring the task energy consumption can misdirect energy reduction techniques such as voltage and frequency scaling or dynamic power management. Focusing on the FPGA, we categorise the misconceptions into four pitfalls that are explained below.

Pitfall 1: The first pitfall is considering the baseline or background power as a part of the task power consumption. As the software tasks are running on the processing subsystem, one may include the PS or MEM baseline power or other task (i.e., background power) in the task power. This happens when only one application is considered for comparison between two different FPGA and CPU implementations such as [14] that has considered the CPU background power as the task power. Although, this may be acceptable for platforms dedicated to run a single application, it may lead to a miss-energy-management in platforms which tend to run multiple applications, simultaneously. Considering the baseline or background power in the software task increases its power consumption compared to the FPGA and makes the FPGA implementation the low power option in most cases which may not be true.

Pitfall 2: The second pitfall consists of ignoring the power added to memory subsystem (or other cores involved in the task execution) due to running the task. This is common in literature [1, 14, 15] and it can be true when different implantations read the same amount of data with the same access pattern from the memory. However, it can mislead the energy evaluation process, if different implantations have different cache mechanism or storing local data which is the case in FPGA. Ignoring this power component underestimates the task power and exaggerates the impact of energy reduction techniques such as FPGA voltage and frequency scaling which does not have an impact on the memory subsystem. This can be clarified by extending the Amdahl's law to energy consumption by which the part of the task energy consumption that does not change by the FPGA voltage and frequency scaling diminish the total task energy reduction.

Pitfall 3: Ignoring the FPGA idle power is the third pitfall. This happens when only one iteration of a periodic task is used for energy evaluation or when it is assumed that FPGA always runs a task without idle time which may not be true in real cases, such as in [14]. As the FPGA-based accelerator power cannot be a part of the baseline power, we should consider the FPGA energy consumption while it is idle as the hardware task power. Otherwise, the measured hardware task power can be much less than the real power which may misconduct the selection process between hardware and software tasks to reduce the energy consumption. Studying the FPGA idle power requires more reasoning as the FPGA idle time can be changed by VFS and makes the FPGA energy efficient in some cases. In addition, this idle energy can be reduced by putting the PL in sleep mode in which the clock is gated and the FPGA voltage is reduced to a level higher than the *data retention voltage* [8] below which the FPGA loses its configuration.

Pitfall 4: The last pitfall is ignoring the runtime system behaviour which may impose significant overhead on the accelerator energy consumption. This mainly happens when the overheads of switching between different implementations are ignored such as in [15]. Note that, the runtime behaviour of the workload and the variable performance required by the system to run multiple tasks can have a negative impact on energy saving techniques. Two practical cases will be explained briefly in the sequel.

Case 1: In this case, the task deadline or period reduces. The task deadline reduction may eliminate some of the software implementations due to the timing constraints, and leave the FPGA implementation the only option which dictates the energy consumption. In addition, the task period reduction, may prevent putting the idle FPGA into the sleep mode as switching between sleep mode and active mode is associated with some timing overhead that may be longer than the FPGA idle time.

Case 2: In this case, the task period and deadline increases. This case increases the FPGA idle time and consequently FPGA energy consumption. One option to reduce the FPGA energy is shutting down the FPGA if the idle time is long enough to cover the timing and energy overhead caused by FPGA reconfiguration.

3.2 Motivation example

Considering an instructive task example, this subsection quantitatively shows the impact of each pitfall. Let's consider an application (called A) which contains the matrix-vector multiplication (mxv) as a periodic task which in each iteration multiplies a 2000×2000 matrix by a vector of length 2000. For the sake of simplicity, other tasks in the application A are modelled by a task called S. The processor runs task S, however, task mxv can be run on the FPGA or the processing system. In addition, a data dependency is assumed between mxv and S that dictates a sequential execution between them as shown in the timing diagram of Fig. 4. Columns 2-7 of Table 1 show the execution time, power and energy consumption of one iteration of task mxv running on the single-core, dual-core, dual-core+NEON and the FPGA in the Zynq SoC. According to Fig 3, the software tasks cause power consumption on the PS and MEM subsystem and the task running on the FPGA adds power on the PL, MEM and PS. For the purpose of comparison, the FPGA power in Column 6 represents its active energy while running the task, and the total FPGA energy including active and idle period is shown in Column 7. Note that the power in this table is the power of running tasks added to the system and does not include the baseline power. The baseline power, which is shown in the last line of this table, are about $0.318W$ and $0.575W$ for the PS and memory subsystem in the Zync SoC running Ubuntu Linux, respectively.

Pitfall 1: The eighth column in Table 1 shows the energy consumption resulting from the first pitfall in which the task power measurements include the PS and MEM baseline power which is about $0.318 + 0.575 = 0.893W$. As can be seen, in this case there is a big difference between the hardware and software tasks energy consumption which is not correct. Note that one may not include the processor baseline power in the FPGA task power which makes the situation even worse.

Pitfall 2: The second pitfall is ignoring the memory (or other cores) power consumption in computing the task energy consumption. As shown in Column 9 of Table 1, this pitfall makes the FPGA implementation more energy efficient while it is not true based on Columns 6 and 7.

Pitfall 3: If we consider the FPGA's idle state, then the energy consumption of the task mapped on the FPGA increases, significantly. In the diagram of Fig. 4, the total period is $50ms$ of which $10ms$ is assigned to task S and $40ms$ is left for mxv. Therefore, the dual-core, dual-core+NEON and the FPGA version of the task mxv meet the time constraint and can be used for the execution.

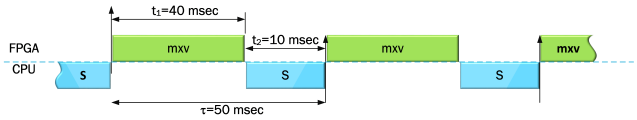
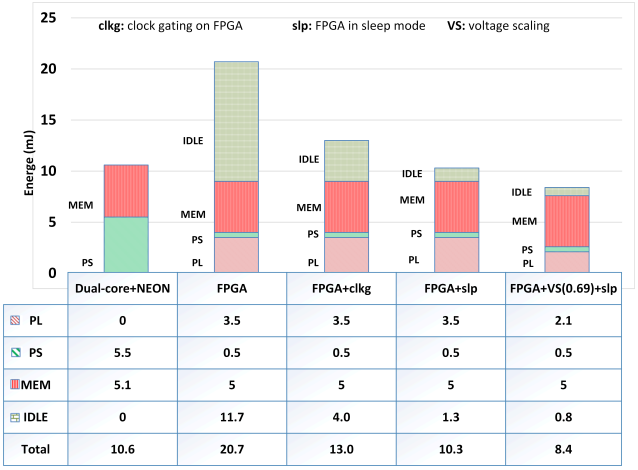


Fig. 4. Motivation example timing diagram

Table 1. Different versions of *mxv* in active model

resources	exe. (ms)	power (W)			task energy (mJ)		energy (mJ)			
		PS	MEM	PL	PL_idle=0	PL_idle=10	Pitfall 1	Pitfall 2	Pitfall 3	Pitfall 4
Single core	53.32	0.08	0.07	0	8.0	8.0	55.61	4.27	8.0	8.0
Dual core	37.31	0.17	0.13	0	11.2	11.2	44.51	6.34	11.2	11.2
dual core+NEON	24.54	0.22	0.21	0	10.6	10.6	32.51	5.40	10.6	10.6
FPGA	6.76	0.07	0.73	active=0.53 idle=0.27	9.0	20.7	15.03	4.06	9.0	11.6
baseline power	—	0.318	0.575	—	—	—	—	—	—	—



	Dual-core+NEON	FPGA	FPGA+clkg	FPGA+slp	FPGA+VS(0.69)+slp
PL	0	3.5	3.5	3.5	2.1
PS	5.5	0.5	0.5	0.5	0.5
MEM	5.1	5	5	5	5
IDLE	0	11.7	4.0	1.3	0.8
Total	10.6	20.7	13.0	10.3	8.4

Fig. 5. Energy consumption (mJ)

However, choosing the proper version for reducing the energy consumption requires more analysis. Fig. 5 compares the FPGA implementation with the fastest software version satisfying the timing constraint, which is the dual-core+NEON version. The PS and MEM consume $5.5mJ$ and $5.1mJ$, respectively, to perform the software version. On the other hand, the FPGA implementation energy consumption has four sources: the PL when it is active, the PL when it is idle, the PS and MEM. Therefore, the total energy consumption added to the system by the hardware implementation is about $20.7mJ$ which is much higher than that of the corresponding software implementation (i.e., $10.6mJ$). Note that, in this case, the idle energy consumption is the dominant part in the FPGA implementation. This part can be reduced by clock gating the PL. The energy consumption of the task *mxv* considering PL clock gating during the idle time is denoted by FPGA+clkg in Fig. 5. The PL idle energy can be reduced further by putting the PL in sleep mode in which the clock is gated and the PL voltage is reduced to 0.6 V that is slightly higher than the data retention

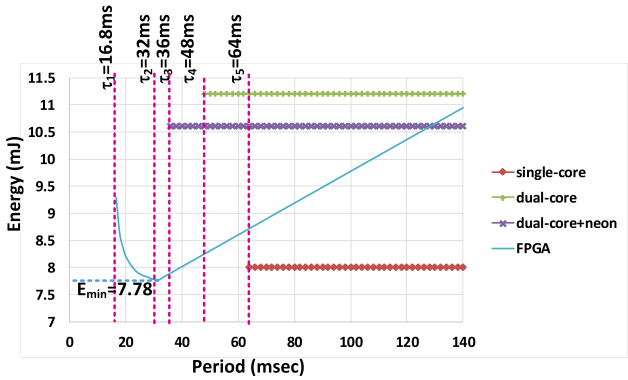


Fig. 6. Energy versus period (*mxv*)

voltage below which the FPGA loses its configuration. This case is denoted by FPGA+slp in Fig. 5. Putting the FPGA in sleep mode reduces the total energy consumption of the task to $10.3mJ$ which is slightly less than that of the software implementation. However, in this case there is $40 - 6.76 = 33.24ms$ slack in the task *mxv* timing, so scaling down the PL voltage while it is active further reduces the total energy consumption. Reducing the PL voltage to $0.69mv$ increases the *mxv* execution time to $21.01ms$ and reduces the PL and MEM average powers to $0.10W$ and $0.2W$, respectively. Note that, whereas voltage scaling directly reduces the PL power consumption, frequency scaling reduces the MEM power consumption because of the reduction in memory access frequency. The FPGA idle power in this case is equal to $0.0216W$, therefore the energy consumption to $(0.07 + 0.2 + 0.1) * 21.01 + 0.0216 * (50 - 21.01) = 8.4mJ$ which is about 20.75% more efficient than the software implementation. This case is denoted by FPGA+vs+slp in Fig. 5

Pitfall 4: As an example of this case, if the *mxv* task period increases to $200ms$ because of some changes in the required input performance, the energy consumption of the hardware implementation with voltage scaling and sleep mode increases to $(0.07+0.2+0.1)*21.01+0.0216*(200-21.01) = 11.6mJ$ which is higher than the energy consumption of the software implementation. Therefore, without considering dynamic workload, choosing the FPGA as the task implementation platform leads to a higher energy consumption.

To summarise the above discussion, Fig. 6 shows the energy consumption of different implementations versus period to clarify the trade-off between the energy consumption and performance. Note that as mentioned before, this paper only considers the VFS in the FPGA. However, considering the VFS in processing system or even the main memory complements this research and all methods presented in this paper are still applicable. For the period between τ_1 and τ_3 in Fig. 6, only the FPGA implementation is acceptable due to the timing constraint. During τ_1 to τ_2 the voltage and frequency scaling can be applied to the FPGA to reduce the energy consumption. Beyond the τ_2 time instance the VFS cannot scale down the FPGA's voltage further, as it reaches its minimum value. Therefore, after this point the FPGA idle energy increases the task energy consumption with a constant rate. At the τ_3, the dual-core+NEON implementation can satisfy the timing constant and it would be one of the options for executing the task *mxv*, however its energy consumption is higher than that of the FPGA. At the time instance τ_4 the dual-core implementation can be another choice but its energy consumption is quite high. The single-core implementation can be used beyond τ_5 point, which also provides the minimum energy consumption.

3.3 Definitions and problem formulation

We assume an embedded system consists of processors, main-memory and an FPGA. The processors run an operating system managing the whole system and running applications. Let's assume application A contains a periodic task s which has a period and a deadline denoted by τ and δ, respectively, such that $\tau > \delta$. The period and deadline of a task is called its *state* denoted by $\mathbb{S}$. The amount of time that the task s stays in its state, which is called *state life-time* and denoted by $T(\mathbb{S})$, is another factor that characterises the task. Task s has at least two software and hardware implementations with different speed and energy characteristics. Our main goal is finding the best task implementation that satisfies deadline constraints and minimizes the consumed energy. For this purpose, two problems , denoted by *P1* and *P2*, are addressed in this paper. Note that, these two problems complement each other, hence, studying one without another reduces their application in real systems.

P1: The first problem is finding the best implementation for a periodic task with a known state $\mathbb{S}$ to minimise the energy consumption considering VFS in the FPGA as well as the FPGA shut-down. The outcome of solving this problem can be one of these three cases:

- Using the FPGA implementation with VFS and putting it into the sleep mode for the rest of the period
- Using the FPGA implementation with VFS and then turning it off for the rest of the period
- Using a software implementation

Note that because the first three pitfalls deal with the energy measurement, this problem will address them in its solution.

P2: Considering the dynamic behaviour of task state, the result of P1 and the overhead of switching between the two different implementations, the second problem determines when switching between two implementations saves energy. This problem copes with the last pitfall in which the state of a task may change during its execution.

Table 2 formally defines the first problem. The first line says that task s has n different software (represented by $soft_i$) and an m FPGA-based (denoted by $accel_i$) implementations. Line 2 determines the task state. Lines 3 and 4 consider that the execution time and energy consumption of software versions are known and do not change. Lines 6-9 show the relation among energy, frequency, voltage of the FPGA implementations under VFS scenario. The functions h_i, e_i and g_i will be determined in the rest of this paper. Lines 10 and 11 represents the timing and energy overheads associated with the FPGA turn-on/off which mainly comprise of the amount of time and energy required for the FPGA full reconfiguration. We assume that the deadline is the only timing constraint which is denoted in Line 12. Eventually, Line 13 clarifies that the goal of the problem is task energy optimisation. Corresponding to the single core, dual-core, vector processor and FPGA in the Zynq-SoC, throughout this paper we consider three software and one hardware implementations, that is $n = 3, m = 1$.

If the period and deadline are constant during the task execution, then the problem shown in Table 2 can be used alone to optimise the task energy consumption. However, the task deadline and period can change dynamically at runtime because of changes in the workload and the frame rate of the input. Therefore, dynamically utilising the problem in Table 2 would be the goal of P2 shown in Table 3. The assumptions of this problem are the new state of the task s (Line 1), the function q which determines the life-time of the new state (Line 2), the current implementation corresponding to the previous state of s (Line 3), the new and energy efficient implementation of the task corresponding to its new state resulted from P1 (Line 4) and the overhead caused by the switching between the two implementations (Line 5).

Table 2. P1 formulation

Assumptions:	
1	$s \in \{soft_1, ..., soft_n, accel_1, ..., accel_m\}$
2	τ and δ: period and deadline of task s
3	for all $soft_i$ implementations
4	t_{task} and E_{task} are constant
5	for each $accel_i$ implementation
6	$E_{task} = h_i(V_{fpga})$
7	$t_{task} = e_i(f_{fpga})$
8	$V_{min} < V_{fpga} < V_{max}$
9	$V_{fpga} = g_i(f_{fpga})$
10	$t_{on/off}$: FPGA on/off timing overhead
11	$E_{on/off}$: FPGA on/off energy overhead
Timing constraint:	
12	$t_{task} < \delta$
Goal:	
13	minimize E_{task}
Output:	
14	$impl_{opt}(s)$ the best implementation of s

Table 3. P2 formulation

Assumptions:	
1	$\mathbb{S}$: new state of task s
2	$T(\mathbb{S}) = q(\mathbb{S})$
3	$impl_{curr}(s)$ current implementation of s
4	$impl_{opt}(s)$ the output of P1
5	sw_{to}, sw_{eo}: timing and energy overhead of switching between $impl_{curr}(s)$ and $impl_{opt}(s)$
Goal:	
6	minimize E_{system}

4 PREVIOUS WORK

Considering the define problems, firstly, this section concisely reviews the related research in the literature and then it points out the contributions of this paper and its differentiation from other research. Whereas, DVFS and DPM are well known techniques in processor-based computing systems, there are only a few research attempts to study their behaviour applied to FPGA [1, 6–8, 16, 17]. Among them, [1, 6, 7] study DVFS applied to the FPGA. However, they have not considered the energy consumption caused by the FPGA idle time as well as the main memory energy consumption. This restricts their studies and the applicability of their results. In contrast, we will consider the FPGA idle periods as well as the memory subsystem energy consumption and will show that for some tasks the main memory can drastically restrict the applicability of DVFS on the FPGA. A run-time power gating technique for embedded FPGA is presented by [8]. The paper shows that although FPGA power gating causes timing and energy overheads, it can be used under some conditions to save energy. A utilisation of the FPGA power gating is presented in [17] for streaming applications. Yang et. al [14] compare the FPGA with the processing system energy consumption for two image processing applications. They also propose a regression based energy model for the system to be used at runtime for implementing the DVFS on the FPGA and the processing system. However, they have not considered the memory energy consumption for the edge detection and blurring, the two image processing tasks considered in their research, which are traditionally memory-intensive applications. In contrast to their approach, our energy model in this paper includes the memory energy consumption.

5 ENERGY MODEL GENERATION

This section explains Phase 2 of the dynamic energy management flow shown in Fig. 1 which is the energy model generation. Considering three examples with different burst memory access patterns, this subsection empirically studies the relation between the task energy consumption and the FPGA voltage. The three tasks considered in this section are an edge detection algorithm called Sobel filter (*sobel*), Black-Scholes (*bs*) option pricing and n-body problem (*nb*) [10], which their burst memory access patterns are shown in Fig. 7.

The *sobel* task detects edges in an HD image of size 1920×1080. Using four non-cached high-performance memory ports with separate read and write channels available in the Zynq, the design reads four image pixels per clock in a streaming manner, applies the computation to the received pixels and writes back the four result pixels in the memory. This read/write pattern is shown in

Fig. 7a in which the memory is accessed along the task execution. The memory controller multiplexes all the eight read and write channels into one high-speed channel to access the main memory [18]. This causes high switching activities on the data and address buses which increases the power and energy consumptions. In each clock cycle it reads/writes four pixels from/into memory which results in $1.3GByte/sec$ bandwidth utilisation at $f = 100MHz$. As the total bandwidth provided by four 32-bit memory port considered in this design is $4*4*(100MHz) = 1.6GBte/sec$, the bandwidth utilisation performance in this design is $1.3/1.6 = 81.25\%$.

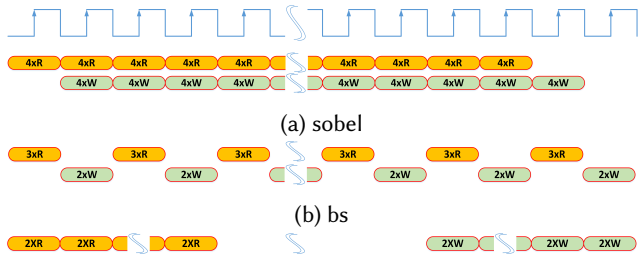
The *bs* task applies the Black-Scholes model to a group of options. The implementation benefits from data streaming and computational pipelining. However, as the task contains several complex mathematical functions such as *log* and *exp*, the resulted implementation reads/writes data from/to memory in every other clock cycle. Fig, 7b shows the corresponding burst memory access pattern during which the accelerator reads three floating point data and writes two floating point data, alternatively. The experimental bandwidth utilisation of this design is $0.749GByte/sec$ at $100MHz$ frequency, therefore, its bandwidth utilisation performance is 46.8%.

The last task is the n-body problem (*nb*) with 4096 particles that consists of two nested loops. The inner-loop requires to have access to all input data to generate one output. Therefore, the accelerator reads all the required data into the FPGA internal memory and after performing the computation the results are written into the memory. Fig. 7c depicts this burst memory access pattern. The experimental bandwidth utilisation of this design is $0.0028GByte/sec$ at $100MHz$ frequency, therefore, its bandwidth utilisation performance is 0.175%. The main reason of this low memory utilisation is that the task is compute-intensive.

Considering the three aforementioned tasks. Fig. 8a, Fig. 8b and Fig. 8c show the task energy consumption versus the FPGA voltage levels for different components involved in the execution which are the PS, PL and MEM. As can be seen, the MEM energy consumption is dominant in the *sobel* filter as its pipelined streaming implementation accesses the memory in each clock cycle for reading and writing data. The MEM energy consumption in the *bs* has a great contribution to the total energy consumption as the memory is accessed for reading and writing data in every other clock cycle. The *nb* task shows a negligible MEM energy consumption as it is compute-intensive and has limited amount of memory transaction at the start and end of task execution. The FPGA energy consumption is mainly varied with the voltage. However, negligible changes in MEM and PS energy consumptions can be seen in diagrams in Fig. 8. This is the impact of the small logic circuit interfaces that exist between different domains. The PS domain has a small logic circuit that generates the PL frequency. Any changes in the generated frequency, will change the switching speed on part of this circuit which results in changes in energy. Similar reasoning can be done for the memory subsystem. As mentioned before in this section, a logic circuit in the memory controller multiplexes separate read/write channels of the FPGA into a single read/write channel of the external memory. Any changes on the FPGA memory access speed, via FPGA frequency, has a slight impact on the memory subsystem energy consumption.

As the total task energy consumption is the sum of three energy components, i.e., $E_{total} = E_{pl} + E_{mem} + E_{ps}$, and only E_{pl} is a function of the FPGA voltage, then the sensitivity of the total energy to the FPGA voltage i.e., $\partial E_{total}/\partial V$ is equal to $\rho = \partial E_{pl}/\partial V$. This factor represents the rate of saving energy by reducing the FPGA voltage. Based on Figs. 8a, 8b and 8c, these sensitivities are 3, 107.9 and 29.7 for *sobel*, *bs* and *nb*, respectively.

The memory energy consumption has negative impact on the VFS efficiency which is defined as the percentage of the task energy saving by applying VFS. As the task energy is equal to $E_{task} = E_{pl} + E_{mem} + E_{ps}$, and applying the VFS to the FPGA reduces the E_{pl} by a factor of ρ then the new energy would be $E'_{task} = \rho E_{pl} + E_{mem} + E_{ps}$. According to this equation, the amount of

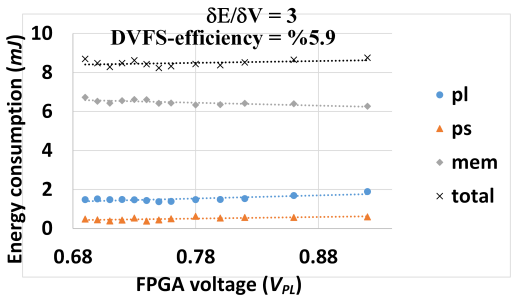


(a) sobel

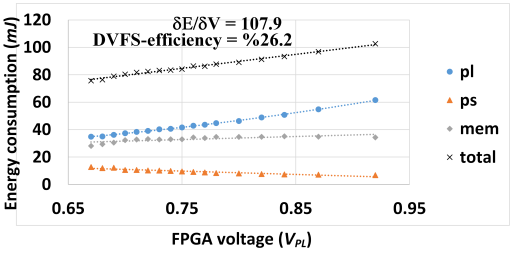
(b) bs

(c) nbody

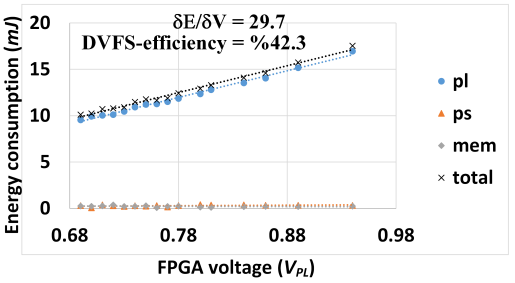
Fig. 7. Burst memory access pattern



(a) sobel



(b) bs



(c) nbody

Fig. 8. Tasks energy consumption

memory and CPU energy consumptions restrict the percentage of the total energy saving. This is similar to Amdahl's law in which the sequential part of a program diminishes the total speed off obtained by the parallel part. Therefore, high memory energy consumption can reduce the impact of VFS efficiency i.e., $(E_{task} - E'_{task})/E_{task}$. This is the reason that the sensitivity of energy consumption in the *bs* task to the voltage is 107.9 which is more than that of the *nb* task (i.e., 29.7), but its VFS efficiency is 26.2% which is less than that of *nb* (that is 42.3%).

According to Figs. 8a, 8b and 8c, there is a linear relation between the energy of the task running on the FPGA and the voltage. Equ. 1 models this linear relation in which α_1 and α_2 are constant that can be obtained by applying a linear regression technique on a set of empirical data for a given

task.

$$E^{task}_{FPGA_{active}}(V) = \alpha_1 V + \alpha_2 \tag{1}$$

This relation can also be justified, analytically, with the following simplified discussion. The active power of the task running on the FPGA consists of three main components: FPGA, memory and processor powers as formulated in Equ. 2. The FPGA power consists of dymaic and static components and are functions of frequency and voltage. The task memory power is a function of frequency as a design with higher frequency performs the read and write operation with higher speed which increases the memory dynamic power. Finally, the processor power is constant as it just sends or receives a few arguments to the design before or after task execution. As shown before, the PS power is negligible and if we ignore the FPGA static power compared to its dynamic, then Equ. 3 approximates the power consumption. Considering the relation between frequency and the execution time in FPGA as Equ. 4, then Equ. 5 approximates the energy consumption. As in new silicon technologies the range of voltage scaling (i.e., (V_{max}, V_{min})) is small, $E^{task}_{FPGA_{active}}$ versus voltage can be approximated by a linear relation using the Taylor series expansion [19] as shown in Equ. 6 in which $V_{min} < b < V_{max}$. Equ. 7 shows the amount of error caused by this approximation in which $V_{min} < \xi < V_{max}$. Subsection 8.3 demonstrates the error of the energy model based on this approximation and compares that with the models derived from the analytical formulation of Equ. 2 for thirteen different benchmarks.

The FPGA idle energy is equal to multiplication of FPGA idle time and power as shown in Equ. 9, in which T represents the task's period.

$$P^{task}_{FPGA_{active}} = \overbrace{\underbrace{a_1.f.V^2}_{dynamic\ power} + \underbrace{a_2.V}_{static\ power}}^{FPGA\ power} + \overbrace{a_3.f}^{MEM\ power} + \overbrace{a_4}^{PS\ power} \tag{2}$$

$$P^{task}_{FPGA_{active}} \approx a_1.f.V^2 + a_3.f \tag{3}$$

$$t_{task} = \mathbf{e}(f) = \theta/f \tag{4}$$

Substituting Equ. 1 and Equ. 9 into Equ. 8, Equation 10 represents the FPGA energy model. This model can be further simplified by considering the relation between the execution time and frequency shown in Equ. 4 and the linear relation between the FPGA voltage and frequency under VFS [20] which is represented as Equ. 12. Equ. 13 represents the FPGA energy model versus voltage after substituting teh frequency f from Equ. 12 into Equ. 13. Note that α_1, α_2, β_1, β_2 and θ can be found by using linear regression which then they will determine the coefficients in Equ. 13. In summary, the proposed DEM approach uses the three equations $\mathbf{e}(f)$, $\mathbf{g}(f)$ and $\mathbf{h}(V)$. As the linear regression learning technique is the main approach to find coefficients of these equations, a set of experimental measurements by running each task and changing f and V as two independent variables is required. The measurement process monitors the power consumption of PS, MEM and FPGA as well as the execution time of the task running on the FPGA. The $\mathbf{e}(f)$ and $\mathbf{g}(f)$ are determined by directly learning the coefficients θ, β_1 and β_2 from the data set. The coefficients α_1 and α_2 in Equ. 1 are also determined by regression algorithm, then they are used in Equ. 13. Note that, P^{FPGA}_{sleep} is also a constant value which is the FPGA power when it is in the sleep mode. Subsection 8.3 evaluates the accuracy of this modelling technique considering different benchmarks.

$$E^{task}_{FPGA_{active}} \approx a_1.\theta.V^2 + a_3.\theta = u(V) \tag{5}$$

$$E^{task}_{FPGA_{active}} \approx u(b) + \frac{\partial u}{\partial V}(b).(V-b) \tag{6}$$

$$R = \frac{1}{2!}\frac{\partial^2 u}{\partial V^2}(\xi).(V-b)^2 \tag{7}$$

$$E^{task} = E^{task}_{FPGA_{active}} + E_{FPGA_{idle}} \tag{8}$$

$$E_{FPGA_{idle}} = (T - t_{task}).P^{FPGA}_{sleep} \tag{9}$$

$$E^{task} = \alpha_1 V + \alpha_2 + (T - t_{task}).P^{FPGA}_{sleep} \tag{10}$$

$$E^{task} = \alpha_1 V + \alpha_2 + P^{FPGA}_{sleep}.T - \theta.P^{FPGA}_{sleep}/f \tag{11}$$

$$V = \mathbf{g}(f) = \beta_1.f + \beta_2; V_{min} < V < V_{max} \tag{12}$$

$$E^{task}(V,T) = \mathbf{h}(v) = \alpha_1 V + \alpha_2 + P^{FPGA}_{sleep}.T - \theta.P^{FPGA}_{sleep}.\beta_1/(V-\beta 2) \tag{13}$$

6 DYNAMIC ENERGY MANAGEMENT

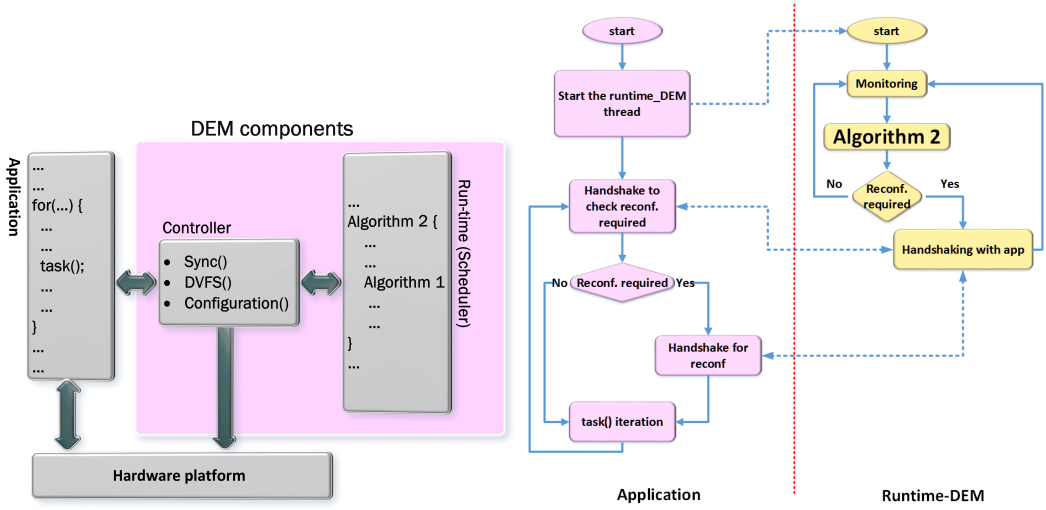
Phase 3 of the dynamic energy management flow, shown in Fig. 1, is explained in this section. An overview of the whole system structure is shown in Fig. 9a. It consists of two main parts, the *runtime system* that monitors the system energy and finds the best implementation for a given task and the *controller APIs* library which consists of a few functions for synchronisation between the application and the runtime system, performing the voltage/frequency scaling and FPGA configuration. The flowchart of the runtime system and its interaction with thr application is shown in Fig. 9b. The runtime system is invoked by the application and performs all the configuration with the permission of the application as the periodic task should be in a proper state in order to prevent malfunctions. In our implementation, when an application starts, it creates a thread running the runtime-DEM which gets the task states and finds the best implementation for the given task using two algorithms to cope with the two problems introduced in Subsection 3.3. Note that, the runtime system calls the algorithm that solves the P2 problem which it in turn calls the algorithm dealing with the P1 problem. These algorithms are explained in the sequel.

6.1 Problem P1 algorithm

As mentioned in Subsection 3.3, the main goal of Problem P1, shown in Table 2, is finding the most energy efficient implementation for a given periodic task. Equs. 4 to 13 provide all the models we need to solve this problem. Using these models, we calculate two factors that will be used to find the best implementation. The first factor which is called *transition point* factor (t_{tp}) determines the minimum period at which the FPGA implementation (with VFS) consumes less energy than a software version. This can be obtained by solving Inequ. 14, in which $E^{task}(V, \tau_{tp})$ is based on the Equ. 13 and $E_{software}$ is the minimum energy of the software implementation that satisfies the timing constraints.

$$E^{task}(V, \tau_{tp}) \leq E_{software} \tag{14}$$

There can be one τ_{tp} for each software implementation. In this case, τ_{tp} factors that can be listed in an ascending order whose corresponding energy consumptions are in a descending order are



(a) Component overview (b) DEM flowchart

Fig. 9. Dynamic Energy Management (DEM)

considered. This justifies that by increasing the task period there exists a different energy efficient implementation. For the sake of simplicity, in our discussion, we consider the minimum τ_{tp} in this list.

The second factor which is called *on-off point* factor (τ_{ofp}) determines the point at which switching off the FPGA during its idle time helps the FPGA implementation consumes less energy than the software version. This factor can be obtained by solving Inequ. 15 and 16, in which $E^{task}_{FPGA_{active}}(V)$ and t_{task} are based on Equ. 1 and Equ. 4, respectively, and the energy and timing switching overheads are denoted by $E_{on/off}$ and t_{ofp}, respectively.

$$E^{task}_{FPGA_{active}}(V) + E_{on/off} \leq E_{software} \tag{15}$$

$$t_{task} + t_{ofp} \leq \tau_{ofp} \tag{16}$$

Note that if these inequalities do not have a solution, for example, if the FPGA on/off energy overhead (which mainly caused by the FPGA reconfiguration and is about $9.4mJ$ [8]) is greater than the software energy consumption, that is, $E_{software} < E_{on/off}$, then we assume $\tau_{ofp} = \infty$. Corresponding to each software implementation there can be a τ_{ofp}. For the sake of brevity, we just consider one τ_{ofp} factor in the following discussion.

Considering one τ_{tp} and τ_{ofp}, two cases can happen which are explained in the sequel.

Case 1: $\tau_{tp} < \tau_{ofp}$: In this case, based on the current task period (i.e., τ), three options are possible that are shown in Expression 17. The first option in which $\tau < \tau_{tp}$ denotes that the FPGA implementation along with VFS can consume the minimum energy. If the task period is less than the *on-off point* factor but greater than the *transition point* factor (i.e., the second option in Expression 17) then the software implementation consumes less energy. Finally, if the task period is greater than the *on-off point* factor, then the FPGA implementation along with VFS and on/off would be the best choice for reducing the energy.

$$\tau_{tp} < \tau_{ofp} : \begin{cases} FPGA + VFS & if \quad \tau < \tau_{tp} \\ Software & if \quad \tau_{tp} < \tau < \tau_{ofp} \\ FPGA + VFS + on/off & if \quad \tau > \tau_{ofp} \end{cases} \tag{17}$$

Case 2: $\tau_{tp} > \tau_{ofp}$: In this case as the τ_{ofp} is less than τ_{tp}, then for the task period less than τ_{ofp} the FPGA implementation with VFS is more energy efficient and for the task period greater than τ_{ofp}, switching off the FPGA is the option. After this point, the task energy does not increase as FPGA is off, therefore the software option never gets a chance to be more energy efficient. Expression 18 denotes this case.

$$\tau_{tp} > \tau_{ofp} : \begin{cases} FPGA + VFS & if \quad \tau < \tau_{ofp} \\ FPGA + VFS + on/off & if \quad \tau > \tau_{ofp} \end{cases} \tag{18}$$

Algorithm 1, which receives the task state (i.e., its deadline and period), the *transition point* and the *on-off point* factors, attempts to find the best implementation for the given task to minimise the energy by investigating different intervals explained in Expressions 17 and 18. Line 1 represents the ascending ordered list, denoted by τ_p, of all point factors (i.e., τ_{tp_i} and τ_{ofp_i}) whose energy consumption are in a descending order. The *for* loop at Line 2 traverses over the elements in τ_p to find the best interval for the input task period (i.e., τ). The condition statement in Lines 4 to 12 selects the best FPGA implementation considering the VFS. The software implementation is chosen if the condition statement in Lines 13 to 16 satisfies. Note that, the condition at Line 13 checks the validity of the second condition in Exp. 17, the only case that software implementation is more energy efficient. In this case, the task period should be greater than a transition point factor. As at this point τ, the task period, is less than τ_{pi} and greater than $\tau_{p(i-1)}$, then $\tau_{p(i-1)}$ should be of type software transition point factor. The execution of conditional branch in Lines 16 to 20 selects the FPGA implementation along with turning off the FPGA during its idle time.

6.2 Problem P2 algorithm

This subsection explains a solution to the P2 problem shown in Table 3 that uses Algorithm 1 to optimise the total energy consumption in a system by selecting either the hardware or software implementation of a periodic task in a given application. As assumed in this paper, the processors are running different tasks and should be always alive. Hence, if the hardware implementation is chosen, processors are allowed to spend more time executing other tasks assigned to them. However, if the software implementation is chosen, there are two cases that can be considered for the FPGA.

Case 1: If there is another task eligible for running on the hardware or FPGA hosts multiple tasks, then the FPGA cannot be shut down and should be active running the task. Note that if the new task uses the same FPGA configuration, then the overhead is negligible, otherwise the timing and energy overheads associated with the FPGA reconfiguration should be taken into account. Since, we consider the periodic task, this case leads to interleaving two periodic tasks on the same FPGA which is beyond the scope of this paper and requires a separate articles.

Case 2: If the FPGA does not host a new task, then it would be idle, which means it can be in the sleep mode or can be shut down. Leaving the FPGA in the sleep mode causes energy leak, which can be significant in case of long idle intervals. To tackle this energy leak, turning off the FPGA could be a solution. However, shutting down the FPGA means that the configuration is lost and a full reconfiguration is required for running its task later. This case is considered and explained in the sequel.

ALGORITHM 1: Energy optimization under timing constraint

Data: $\mathbb{S} = \{\delta, T\}$
Data: $\{(\tau_{tp_i}, E_{tp_i})\}$
Data: $\{(\tau_{ofp_i}, E_{ofp_i})\}$
Result: $impl_{opt}$: minimum energy implementation

$\tau_p = (\tau_{p0}, \tau_{p1}, ...\tau_{pn}), \tau_{pi} \in \{\tau_{tp_i}\} \cup \{\tau_{ofp_i}\} \mid \tau_{pi} < \tau_{p(i+1)} \wedge E_{pi} > E_{p(i+1)}$
for $i \leftarrow 0$ **to** n **do**
  if $(\tau < \tau_{pi})$ **then**
    if $(i == 0)$ **then**
      $f_{fpga} = \frac{\theta}{\delta}$
      if $(f_{fpga} < f_{min})$ **then**
        $f_{fpga} = f_{min}$
      end
      $V_{fpga} = \beta_1 . f_{fpga} + \beta_2$
      $impl_{opt} = 0$
      break
    else
      if $(\tau_{p(i-1)} == \tau_{tp_j} \in \{(\tau_{tp_i}, E_{tp_i})\})$ **then**
        $impl_{opt} = j$
        break
      else
        $f_{fpga} = f_{min}$
        $V_{fpga} = V_{min}$
        $impl_{opt} = -1$
        break
      end
    end
  end
end

ALGORITHM 2: Dynamic Energy Management

Result: $impl_{curr}$: current implementation of task s
Data: $\mathbb{S} = \{\delta, T\}$
Data: T_{time_out} : time out policy parameter
Result: sw_{impl}

$sw_{impl} = 0$
$impl_{opt} \leftarrow$ *Algorithm 1*
if $impl_{opt}$ *is hardware AND* $impl_{curr}$ *is software* **then**
  if $fpgaOffFlag == 1$ **then**
    Turn on and reconfigure the FPGA
  end
  sw_{impl} = switch to hardware
end
if $impl_{opt}$ *is software AND* $impl_{curr}$ *is hardware* **then**
  sw_{impl} = switch to software
  fpgaOffFlag = 0;
  Start the timer;
end
if $impl_{curr}$ *is software and the timer is greater than* T_{time_out} **then**
  Shut down the FPGA;
  fpgaOffFlag = 1;
end

Considering the latter case, Fig. 10 illustrates the timing diagram of the switching between hardware and software implementations. Whereas the upper part of this figure shows the host of the task in each timing interval, the lower part illustrates the state of the FPGA in terms of different energy consumption modes. Before time instance t_1 the FPGA is performing the task, however assume that at t_1, the software implementation is more energy efficient because of changes in the task period. This transition from hardware to software occurs at t_2 after the current iteration completed, that is $t_2 - t_1 < \tau$. At this time the process of shutting down the FPGA starts which takes $t_3 - t_2$. The FPGA is off during the time instance t_3 to t_4. Let's assume at time instance t_4 the FPGA is more energy efficient or it is the only option that meets the timing constraint, hence, the task should go back to the FPGA. Consequently, FPGA is turned-on and reconfigured. It is fully operational at t_5 that can execute the task after the current iteration completed in software at time t_6. In summary, during $t_1 \rightarrow t_6$ the FPGA is not running the task and only for the $t_3 \rightarrow t_4$ period, it is shut down; hence, $t_1 \rightarrow t_3$ and $t_4 \rightarrow t_6$ intervals represent the overheads of the DEM.

These overheads restrict the switching between hardware and software implementations and it should take place if the amount of saved energy justifies the overhead. The minimum interval of the idle period to save energy by shutting down the FPGA is called *break-even time*, T_{be}. It depends on

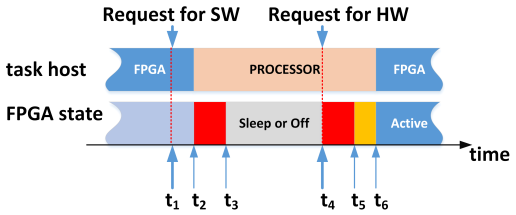


Fig. 10. DEM timing diagram

the FPGA power consumption during the sleep mode and the energy overhead caused by transition from hardware to software implementations and vice versa. Equ. 19 shows the equality of saving energy by turning off the FPGA and the energy required to configure the FPGA.

$$P_{sleep}.T_{be} = E_{switching} \tag{19}$$

According to [8] the timing and energy overhead related to the Zynq SoC reconfiguration are $48msec$ and $9.4mJ$, respectively. In addition, the PL energy power consumption in sleep mode is $0.0293W$. Therefore the break-even point of the Zynq SoC satisfies $0.0293 * T_{be} = 9.4mJ$ and is $T_{be} = 320.8ms$.

Changing the power state of a hardware module at runtime while it is in the idle mode is a well-known technique in Dynamic Power Management (DPM) that reduces the energy in computing systems. In general case, it is not easy to predict how long an FPGA stays in the idle mode in order to predict the potential energy saving resulted from shutting down the FPGA. There are different techniques, known as *policies*, in the literature that cope with this problem and determine whether to shut down the device (here the FPGA) or put it into a sleep mode. These policies can be categorised as: *time-out*, *stochastic*, and *predictive* [20]. This paper considers the time-out policy which is also widely implemented in commercial products[21]. According to this policy, if the FPGA stays for T_{time_out} time in idle mode then it will remain idle at least for T_{be} [20].

Considering the time-out policy, Algorithm 2 shows the implementation of DEM policy. It receives the new task state (i.e., $\mathbb{S}$), the current implementation (i.e., $impl_{curr}$) and the delay (i.e., T_{time_out}) corresponding the time out policy, then, using Algorithm 1 at Line 2, it makes the decision for possible changes in the task implementation. Lines 3 to 8 determine a switch from software to hardware in which based on the value of $fpgaOffFlag$, the FPGA is reconfigured if it is required. The switching from hardware to software take place in Lines 9 to 13. A timer is activated during this process which will be checked in Lines 14 to 17 to implement the time-out policy. If the timer is greater than the T_{time_out}, then the FPGA is turned off.

7 CASE STUDY: ROBOT MAP CREATION

Map creation is one of the periodic tasks in autonomous robots in which a group of sensors regularly senses the environment and then the robot's map creation algorithm builds a map incrementally based on this information. Regularly monitoring the environment with a fixed-period may waste energy especially when the environment does not change rapidly as the robot moves slowly due to obstacles or other events. Therefore, using variable period for scanning the environment can save the energy on sensors and processing systems.

Listing 1 shows the pseudo-code of the simultaneous localization and mapping (SLAM) algorithm based on the genetic algorithms explained in [22]. This code consists of a loop, from Line 3 to

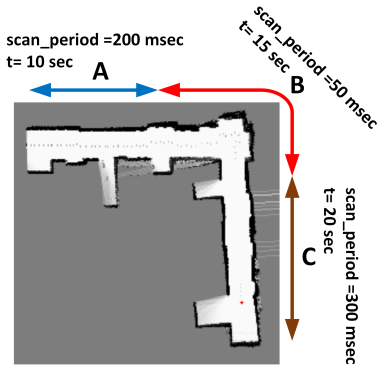


Fig. 11. SMG-SLAM maps examples

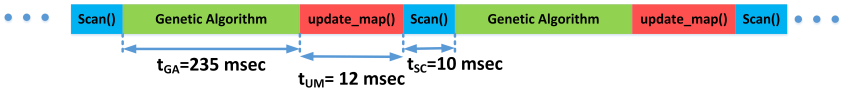
Listing 1. SMG-SLAM map creation pseudo-code

```
1  main() {
2  initialize();
3  while (status = scan()) {
4  //------Genetic Algorithm----------
5  initialise_population();
6  for(int i = 0; i < G ; i++) {
7  update_fitness();
8  fixed_next_generation();
9  }
10 //---------------------------------
11 update_map()
12 }
13 }
```

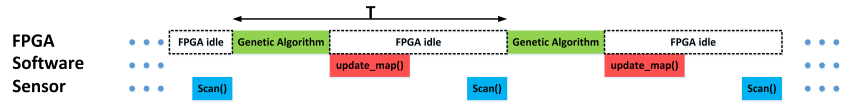
Line 12, that gets the sensor information through *scan()* function and updates the map using a genetic algorithm which is an iterative algorithm represented in Line 5 to Line 9. This algorithm tries to find the best match between the scanned information and the current state of the generated map. After finding the best match, the function *update_map()* at Line 11 updates the map. Fig. 12a depicts the timing diagram of the sequential execution of this application which shows a periodic behaviour. As shown in this figure, the software implementation, using a single core on Zynq, spends 235*msec*, 12*msec* and 10*msec* on iterative genetic algorithm, map update and sensor scan parts, respectively. As the iterative genetic algorithm is compute-intensive, we have implemented it on the Zynq FPGA using the Xilinx Vivado-HLS tool [10]. Table 4 shows the execution time, power and energy consumptions of the genetic algorithm implemented on the FPGA and processors in the Zynq SoC. This table shows that the active energy of the FPGA running the genetic algorithms is less than that of software implementations. However, considering the FPGA idle time and power which are 22*msec* and $0.146W$, respectively, before applying the VFS, the FPGA energy consumption increases to $(6.54 + 22 * 0.146) = 9.75mJ$ which is higher than that of the single core software implementation.

The timing diagram of Fig. 12b shows the execution of the genetic algorithm on an FPGA, the *update_map()* on the processor and the scan on sensors. We assume that the sensors can scan the environment while the processor is updating the map. The timing distance between two consecutive scan defines the task period which should encompass the FPGA task and *update_map* function. A map created by this application is shown in Fig. 11 as examples.

Using the techniques and models presented in Section 5, Equ. 20 to Equ. 23 represent all the models we need for performing the DEM. To demonstrate the effectiveness of the proposed techniques, we divide Path 1 shown in Fig. 11 into three sections. In section A, the robot walks in a straight line with no bend or obstacles and it senses the environment every 200*msec*. Because of a bend along the section B, the robot senses the environment every 50*msec* to quickly detect any possible objects coming from the other side. During section C, it senses the environment every 300*msec* as it realises that this section is the end of the path so it walks slowly which requires less scans. In the sequel, the best implementation for each section will be found.



(a) Sequential timing diagram



(b) FPGA+CPU timing diagram

Fig. 12. SMG-SLAM timing diagram

Table 4. Genetic Algorithm statistics

design	exe. time (msec)	power (W)			Energy (mJ)			
		PL	PS	MEM	PL	PS	MEM	total
FPGA (active)	21.24	0.30	0.006	0.004	6.33	0.116	0.088	6.54
single core	235.94	0	0.037	0.0014	0	8.73	0.336	9.06
Dual core	194.42	0	0.066	0.0019	0	12.8	0.374	13.2
Dual core+NEON	166.45	0	0.072	0.0028	0	11.98	0.457	12.44

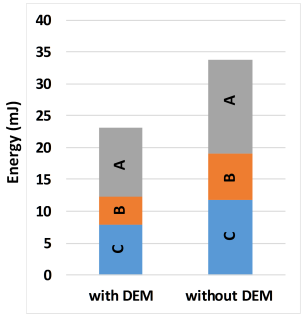


Fig. 13. Energy comparison for different sections

$$E^{task}_{FPGA_{active}} = 12.3V - 5.02 \quad (20)$$

$$t_{task} = \mathbf{e}(f) = 2118/f \quad (21)$$

$$V = \mathbf{g}(f) = 0.0031f + 0.58; 0.7 < V < 1 \quad (22)$$

$$E^{task} = \mathbf{h}(v) = 12.3V - 5.02 + 0.0293.T - 0.19/(V - 0.58) \quad (23)$$

Section A: In this section $T = 200msec$ therefore FPGA task has $T_{GA} < T - t_{UM} = 200 - 12 = 188msec$ time to finish its task. However, the lowest value for FPGA voltage is $0.7v$ which in this case $f = 38.7MHz$. Considering this frequency and Equ. 21, the task execution time is $t = 54.73msec$. Equ. 23 predicts $12.3 * 0.7 - 5.02 + 0.0293200 - 0.19/(0.7 - 0.58) = 7.9mJ$ energy consumption for one scan in the task, which is less than that of the software implementations. Therefore, FPGA is the best option during this section. Note that, considering the FPGA implementation without DVFS consumes $(200 - 21.24) * 0.0293 + 6.54 = 11.77mJ$ energy.

Section B: During this interval, because of the low period (i.e., $50msec$), the FPGA is the only option that satisfies the timing constraint. In this case, the FPGA has at most $50 - 12 = 38msec$ to finish its task. According to Equ. 21 and Equ. 22 the scaled frequency and voltage are $55.73MHz$ and $0.75v$, respectively. Finally, Equ. 23 predicts the energy consumption which is $4.5mJ$. In this case, considering the FPGA implementation without DVFS consumes $(50-21.24)*0.0293+6.54 = 7.38mJ$ energy.

Section C: In this period which $T = 300msec$, similar to the first case, if the FPGA executes the task then $f = 38.7MHz$ and $V = 0.7v$, consequently according to Equ. 23 $E = 10.8mJ$. As can be seen, the FPGA implementation consumes more energy than the single processor software version (which is $9.06mJ$ as shown in Table 4). Therefore, switching from FPGA implementation to the software implementation can save energy. Note that, considering the FPGA implementation without DVFS consumes $(300 - 21.24) * 0.0293 + 6.54 = 14.71mJ$ energy. In summary, Fig. 13 compares the energy consumption in different sections considering two implementations with and without DEM.

If the robot spends $10sec$, $15sec$, and $20sec$ in sections A, B and C, respectively. Then the number of scans in these three sections are 50, 300 and 50, respectively. Considering the energy efficient implementations in each section, explained before, the total energy consumption is $(7.8 * 50) + (4.5 * 300) + (9.06 * 50) = 2193mJ$. However, if the FPGA implementation without considering the DVFS and DEM considered then the total energy consumption would be $(11.77 * 50) + (7.38 * 300) + (14.71 * 50) = 3538mJ$. Therefore, using the proposed techniques in this paper can save 38.0% of energy.

In the above discussion, we have considered that the FPGA is shut down during Section C. However, if we consider a DPM time-out policy in which the FPGA is turned off after a delay, for example $T_{time-out} = 200msec$, then $400 * 0.0293 = 11.72mJ$ energy overhead should be added to the energy consumption during Section C. This reduces the energy efficiency to 37.7%.

8 EXPERIMENTAL RESULTS

Considering several common tasks [10] as our benchmarks and the Xilinx Zynq SoC platform, this section, organised in four subsections, evaluates the proposed techniques. Whereas the first subsection explains the experimental setup, Subsection 8.2 reports the statistics, performance and energy consumption of the tasks. Phase 2 and Phase 3 of the proposed DEM flow shown in Fig. 1 are evaluated in Subsection 8.3.

8.1 Experimental setup

As mentioned throughout this paper, we have used the Xilinx Zynq SoC as a hybrid FPGA-ARM embedded platform. Fig. 5 shows the diagram of our experimental setup. The Xilinx Vivado-HLS and Vivado tool sets are used to synthesise the C++ description of the tasks in our benchmarks. The processor communicates with the task on the FPGA through a few registers to read or write the task arguments. In addition, the FPGA utilises four HP ports for memory transactions. A *start* and an *interrupt* signals are used for handshaking between the task in the FPGA and the processor. An on-board voltage regulator provides all the voltage rails required by the Zynq SoC. This voltage regulator is controlled by the processor thorough the PMBUS protocol. We measured the power consumption of different voltage rails by changing the frequency and voltage of the FPGA for each tasks. A set of MATLAB functions is provided which receives these empirical data to create the proposed energy models using the linear regression algorithm. In addition, we have provided a set of MATLAB functions to implement Algorithms 1 and 2 for evaluation and to compare the results with real measurements in the system. All the codes, libraries and designs used in this paper are open-source and are available at [10].

8.2 Periodic tasks energy behaviour

We have provided three software and one hardware implementations for each task. The software implementations include *serial* which uses only one Cortex-A9 core, *parallel* which uses the dual-core and *vector* which uses the Neon coprocessor as well as the dual-core CPU. Table 6 shows some of the statistics of these benchmarks. The first column represents the name of the task. The

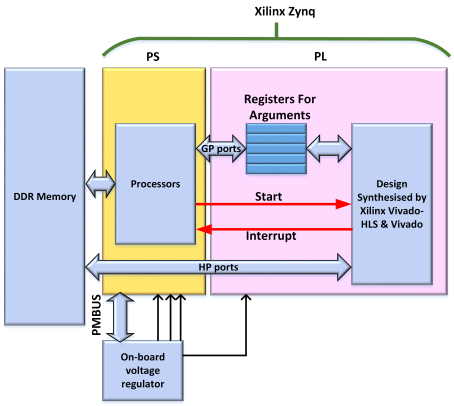
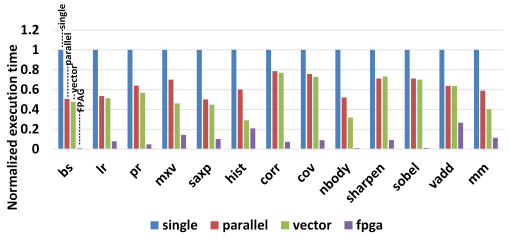


Table 5. Experimental setup

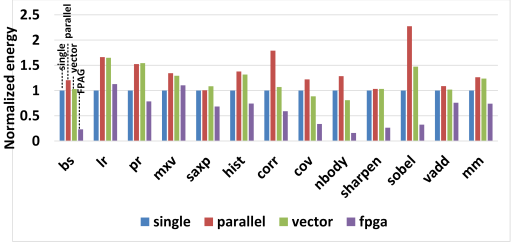
Table 6. Statistics of benchmarks

Name	Complexity		Data (floating point) size (n)	Resource Utilisation		
	Ops	Mem		Slice LUTs %	BRAM %	DSP %
bs	$O(n)$	$O(n)$	4000000	46.13	1.79	91.36
lr	$O(n)$	$O(n)$	16000000	35.31	1.43	26.36
pr	$O(n)$	$O(n)$	32000	47.03	1.43	75
mxv	$O(n^2)$	$O(n^2)$	2000	66.97	13.21	12.27
saxp	$O(n)$	$O(n)$	8388608	33.50	18.57	25.45
hist	$O(n)$	$O(n)$	1048576	8.81	6.43	0
corr.	$O(n^3)$	$O(n^2)$	100	55.10	93.57	62.27
cov.	$O(n^3)$	$O(n^2)$	200	36.85	48.57	54.09
nbody	$O(n^2)$	$O(n)$	4096	63.48	96.43	61.82
sharpen	$O(n^2)$	$O(n^2)$	HD^1	47.34	62.86	43.64
sobel	$O(n^2)$	$O(n^2)$	HD^1	24.42	17.14	3.64
vadd	$O(n)$	$O(n)$	131072	12.08	2.14	1.82
mm	$O(n^2)$	$O(n^2)$	3000	27.94	5.71	5.45

[1] HD : 1080×1920; $n \approx 1440$



(a) Execution time



(b) Energy consumption

Fig. 14. Execution time and energy consumption of benchmarks

interested reader may refer to [10] for the description of each task. The operation and memory access complexities of each task algorithm is shown in the second and third columns, respectively. Column fourth denotes the size of the input data of type float i.e., 4 bytes. The percentage of the hardware utilisations are shown in Columns 5-7 in which Column 5 shows the percentage of Look-Up Table (LUT) slices in the FPGA, Column 6 represents the percentage of Block RAM (BRAM) in the FPGA and the last column shows the utilisation of Digital Signal Processing (DSP) units. Fig. 14a and Fig. 14b compare the execution time and energy consumption of software and hardware implementations normalised to the ones of serial versions, respectively. The FPGA designs provide considerably fast implementations which justifies using the FPGA accelerators under some constraints to speed up the applications and save energy. The FPGA energy consumption in this table is the active energy without VFS which corresponds to the minimum task period. As the software implementations mainly optimised for speed, their energy consumptions depend on their implementations. However, for considered periodic tasks, the serial implementation is more energy efficient.

8.3 Dynamic energy management evaluation

This subsection evaluates Algorithms 1 and 2. Considering two analytical and linear FPGA task energy models explained in Section 5, Fig. 15 compares the maximum percentage of the energy deviation between the result of Algorithm 1 and the ideal minimum average energy (which the

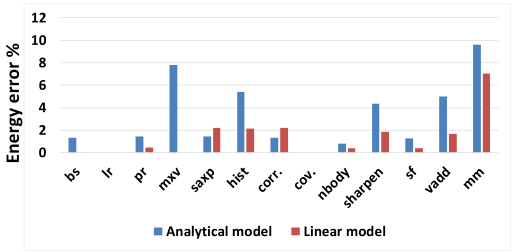


Fig. 15. Maximum energy deviation in Algorithm 1

Table 7. Fast and slow mode deadlines

Name	τ_{tp}	τ_{ofp}	τ_{fast}	τ_{slow}
bs	15000	384.84	–	–
lr	143	∞	71.5	286
pr	341	∞	170	686
mxv	54	∞	27	108
saxp	781	348.25	–	–
hist	407	∞	10	500
corr.	170	∞	10	200
cov.	220	∞	10	300
nbody	3133	344.46	–	–
sharpen	143	339.5	10	200
sobel	566	326.43	–	–
vadd	78	∞	10	100
mm	240	∞	20	300

algorithm should find in the case of using accurate models) obtained by the exhaustive search in measured data. For this purpose, we considered six different periods for tasks and calculated the average errors. As can be seen, both analytical and linear models' errors are less than 10% while the linear model represent less amount of error, which indicates the acceptable accuracy of the proposed linear model for the FPGA active energy and the success of Algorithm 1 to find this minimum.

This subsection evaluates Algorithms 1 and 2. Considering the models represented by Equ. 1 to Equ. 13, Fig. 15 shows the maximum percentage of the energy deviation between the result of Algorithm 1 and the ideal minimum average energy (which the algorithm should find in the case of using accurate models) obtained by the exhaustive search in measured data. For this purpose, we considered six different periods for tasks and calculated the average errors. As can be seen, errors are less than 10% which indicate the acceptable accuracy of the proposed linear model for the FPGA active energy and the success of Algorithm 1 to find this minimum.

To show the effectiveness of Algorithm 2 in action, we consider the tasks for which the switching between two software and hardware implementations is possible as shown in Expression 17. For these tasks, we consider two different modes for the task state to be able to use the two different hardware and software implementations. Whereas, the first mode, called *fast*, requires a high-speed implementation because of the low task period, the second mode, named *slow*, is associated with high task period. For this reason, the period of the fast mode is considered to be less than the *transition point* factor. The slow mode is considered for the software version of the task, therefore, the period of this mode is greater than the *transition point* factor and less than the *on/off point* factor.

Table 7 shows the considered task periods for slow and fast modes as well as the τ_{tp} and τ_{ofp}. Note that, the hardware implementation is always energy efficient in tasks that do not have slow mode period as these tasks satisfy Case 2 denoted by Expression 18.

We have considered three benchmarks in which the two fast and slow modes correspond to the hardware and software implementations, respectively. Fig. 16 shows the normalised consumed energy when these task spends 50% of its periods in the fast mode. The *ideal* values represent the energy consumption without considering the switching overhead. The *proposed* data determines the result of Algorithm 2 considering the switching overhead. It shows that with the time-out DPM policy the impact of switching overhead is negligible. The other values in this figure, denoted by Pitfalls 1 to 4, represent the results of Algorithm 2 when each of the pitfalls described in Subsection 3.1 are not considered. In this diagram, we are studying the impact of each single pitfall on the proposed algorithm. In these designs, as each pitfall fails, it chooses the FPGA implementation

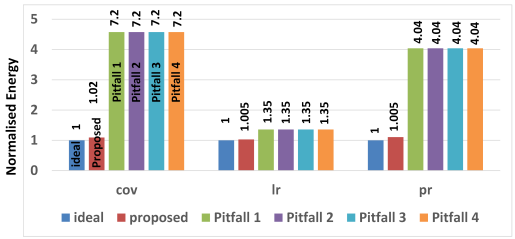


Fig. 16. Pitfalls' energy comparison

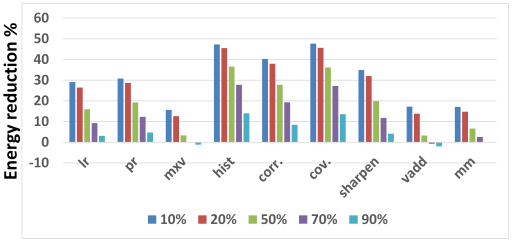


Fig. 17. Time-out policy with $T_{time-out} = 0$

instead of the software one. Therefore, we get the same results for the final energy consumption in Fig. 16.

We have also assumed five different cases in which the switching between software and hardware implementations (i.e., slow and fast modes, respectively) happens with the probability of 10%, 20%, 50%, 70% and 90%. In other words, the task with lower probability spends more time in the slow mode and we expect the lower switching overhead and higher energy saving. Fig. 17 compares the percentage of energy reduction caused by switching between the hardware and software implementations using Algorithm 2 compared to the fully hardware implementation. As can be seen, the amount of energy saving can be more than 48% for applications in which a given task should spend more cycles in the fast mode. Note that, here, we have considered a time-out policy for DPM in which $Ttime\_out = 0$, that is, the task ran by software lasts at least for T_{be}. If this assumption is not met by the task then the switching overhead between hardware and software implementations has negative impact on the total energy consumption. This phenomenon has caused negative the energy reduction percentage for two tasks, i.e. *mxv* and *vadd*, shown in Fig. 17. Note that, the switching between software and hardware implementations causes energy overhead. Therefore, the more switching, the more energy overhead. If the proposed technique can alleviate this energy overhead by properly switching between software and hardware, then it can significantly save the energy. Fig. 17 directly shows this behaviour. If there is more switching, then the algorithm has more change to save energy. This is the main reason of direct relation between the amount of energy reduction and the switching probability.

It is worthwhile to study the scalability and expansion of the proposed energy management technique considering a platform containing multiple CPUs and FPGAs. In this case, the list of transition and on-off point factors increases at Line 1 of Algorithm 1, while the rest of the algorithm remains intact. Moreover, the task transition overheads among FPGAs and CPUs should also be calculated to be used in Algorithm 2.

9 CONCLUSIONS AND FUTURE WORK

This paper has studied the energy consumption of a task running on an FPGA-based accelerator. This study shows that although FPGAs can traditionally provide a fast implementations for different types of tasks, they may not be the most energy efficient solution especially for the memory-intensive applications under a periodic execution scheme. Therefore, a decision made at runtime to run a given task on the hardware or software can significantly save the total task energy consumption. The open-source DVFS framework developed for this paper, benchmarks and the generated data can be found at [10]. The research can be expanded in different ways. One possible avenue of future work studies the energy efficiency of assigning multiple tasks to an FPGA. In addition, scaling the proposed algorithm to cover task distribution among multiple FPGAs and CPUs can be a new line

of research to integrate fine-grained scheduling and binding to the proposed energy management technique.

REFERENCES

[1] J. Luis Nunez-Yanez, M. Hosseinabady, and A. Beldachi. Energy optimization in commercial FPGAs with voltage, frequency and logic scaling. *IEEE Transactions on Computers*, 65(5):1484–1493, May 2016. ISSN 0018-9340.

[2] Intel. Cyclone v device datasheet, 2016.

[3] Intel. Stratix 10 gx/sx device overview, 2016.

[4] S. Zhao, I. Ahmed, A. Khakpour, V. Betz, and O. Trescases. A robust dynamic voltage scaling scheme for fpgas with ir drop compensation. In *2017 IEEE Applied Power Electronics Conference and Exposition (APEC)*, pages 2939–2944, March 2017.

[5] N. Dahir, P. Campos, G. Tempesti, M. Trefzer, and A. Tyrrell. Characterisation of feasibility regions in fpgas under adaptive dvfs. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2015.

[6] Atukem Nabina and Jose Luis Nunez-Yanez. Adaptive voltage scaling in a dynamically reconfigurable FPGA-based platform. *ACM Trans. Reconfigurable Technol. Syst.*, 5(4):20:1–20:22, December 2012. ISSN 1936-7406.

[7] A. F. Beldachi and J. L. Nunez-yanez. Run-time power and performance scaling in 28 nm fpgas. *IET Computers Digital Techniques*, 8(4):178–186, July 2014. ISSN 1751-8601. doi: 10.1049/iet-cdt.2013.0117.

[8] M. Hosseinabady and J. L. Nunez-Yanez. Run-time power gating in hybrid ARM-FPGA devices. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Sept 2014.

[9] System management bus (smbus) specification version 3.0, 2014.

[10] Mohammad Hosseinabady. Dynamic energy management in zynq soc, Accessible 2017. URL https://highlevel-synthesis.com/dynamic-energy-management-in-zynq-soc/.

[11] Mohammad Hosseinabady and Jose Nunez-Yanez. A systematic approach to design and optimise streaming applications on fpga using high-level synthesis. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[12] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 47–56, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1155-7.

[13] A. Ghosh, S. Paul, J. Park, and S. Bhunia. Improving energy efficiency in FPGA through judicious mapping of computation to embedded memory blocks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(6): 1314–1327, 2014. ISSN 1063-8210. doi: 10.1109/TVLSI.2013.2271696.

[14] Sheng Yang, R. A. Shafik, G. V. Merrett, E. Stott, J. M. Levine, J. Davis, and B. M. Al-Hashimi. Adaptive energy minimization of embedded heterogeneous systems using regression-based learning. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on*, pages 103–110, Sept 2015.

[15] Y. Wu, D. S. Nikolopoulos, and R. Woods. Runtime support for adaptive power capping on heterogeneous socs. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 71–78, July 2016.

[16] E Hung, JJ Davis, JM Levine, EA Stott, PYK Cheung, and GA Constantinides. KAPow: A system identification approach to online per-module power estimation in FPGA designs. IEEE. URL http://hdl.handle.net/10044/1/31009.

[17] M. Hosseinabady and J. L. Nunez-Yanez. Energy optimization of FPGA-based stream-oriented computing with power gating. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Sept 2015.

[18] Xilinx Inc. *Zynq-7000 AP SoC Technical Reference Manual.* Xilinx Inc., UG585 (v1.10), February 23, 2015.

[19] Frank W. Olver, Daniel W. Lozier, Ronald F. Boisvert, and Charles W. Clark. *NIST Handbook of Mathematical Functions.* Cambridge University Press, New York, NY, USA, 1st edition, 2010. ISBN 0521140633, 9780521140638.

[20] Yung-Hsiang Lu and G. De Micheli. Comparing system level power management policies. *IEEE Design Test of Computers*, 18(2):10–19, Mar 2001. ISSN 0740-7475.

[21] ACPI advanced configuration & power interface, 2017. URL http://www.uefi.org/acpi/specs.

[22] Grigorios Mingas, Emmanouil Tsardoulias, and Loukas Petrou. An fpga implementation of the smg-slam algorithm. *Microprocess. Microsyst.*, 36(3):190–204, May 2012. ISSN 0141-9331.

Received February 2007; revised March 2017; accepted June 2017